LOIS: technical documentation

Eryk Kopczyński and Szymon Toruńczyk

University of Warsaw, Poland
{erykk,szymtor}@mimuw.edu.pl

Abstract. LOIS is a C++ library allowing iterating through certain infinite sets, in finite time. The resulting language has an intuitive semantics, corresponding to execution of infinitely many threads in parallel. This allows to merge the power of abstract mathematical constructions into imperative programming. Infinite sets are internally represented using first order formulas over some underlying logical structure. To effectively handle such sets, we use and implement SMT solvers for various first order theories. LOIS has applications in education, and in verification of infinite state systems.
This is a technical documentation of LOIS, describing how to write programs using it.

1 Introduction

LOIS (Looping Over Infinite Sets) is a C++ library which allows working on *definable* infinite sets in a natural way. We can create an infinite *domain*, let's say $\mathbb{A}$, possibly with some relational and functional symbols, and then use the *pseudo-parallel* semantics to iterate over it in a natural way. This gives us new sets, for example $\{(x, y) : x \in \mathbb{A}, y \in \mathbb{A}, x \neq y\}$, which can be iterated over in turn, or checked for emptiness. A LOIS program will work in finite time as long as the first order theory of $\mathbb{A}$ is decidable. The current prototype of LOIS is available on the (hidden) webpage:

`http://www.mimuw.edu.pl/~erykk/lois/`

This document is a technical description of LOIS, and thus, many theoretical details have been omitted. See the papers on the website above for details about:

- the theoretical foundations: definable sets, homogeneous structures, [KTa]
- how the solvers (built-in, CVC4, Z3, SPASS) are used to make the computation possible, [KTa]
- the results of our tests of internal and external solvers, [KTa]
- applications, [KTa]
- the novel pseudoparallel semantics which LOIS is using to handle the infinite sets. [KTb]

Note that, in the current prototype, all the core functionality works, but in case of some functions, variants which accept more complex types (say, `rnumof<T>` or `rsetof<T>` instead of the basic `elem` or `rset`) may be still missing, or some obvious type casting might be missing, making C++ unable to guess what typecasts should be used. Usually such more complex variants should be easy to write.

LOIS has been tested on machines with the following configurations:

- gcc version 4.4.3, architecture i686, Ubuntu Linux
- gcc version 4.6.3, architecture x86_64, Ubuntu Linux
- gcc version 4.7.3, architecture x86_64, PLD Linux

2 Tutorial

We start with some examples of simple statements using LOIS (left), and their explanations (right). The key constructs and features are emphasized in the text.

```
#include "../include/
     loisextra.h"

using namespace std;
using namespace lois;
```

To use LOIS, we need to include the appropriate file. LOIS types are in the namespace `lois`.

```
initLois();
sym.useLaTeX();
```

Initialize LOIS. Sets can be output in ASCII (portable), Unicode (useful for testing) or in the LaTeX markup, which we use in this paper.

```
Domain dA("\\mathbb{A}");
lset A = dA.getSet();
cout << "A = " << A << endl;
```

Create an infinite **domain**, named $\mathbb{A}$. The variable `A` stores its underlying set A. This outputs $A = \{a|a{\in}\mathbb{A}\}$.

```
lset X = A;
X += 5;

cout << "X = " << X << endl;
```

Create a **set** X, and set it to A. The type `lset` is the type of a set; `l` signifies that we will use it as an lvalue, that is, change its contents. Sets can store elements of various types. We **add** the number 5 to X. This outputs $X = \{b|b{\in}\mathbb{A};\ 5\}$, a shorthand for $\{b|b \in \mathbb{A}\} \cup \{5\}$.

```
lsetof<elpair> Pairs;

for(elem a: A) for(elem b: A)
    Pairs += elpair(a,b);

cout << "Pairs = "
  << Pairs << endl;
```

The set `Pairs` is a **typed set**, whose elements are of the type `elpair`, denoting pairs; initially it is empty. **Iterate** over all pairs of elements of A, and construct the set of all ordered **pairs**. The output is $Pairs = \{(c,d)|d{\in}\mathbb{A},\ c{\in}\mathbb{A}\}$.

```
lbool ok = true;

for (elem a:A) for (elem b:A)
     for (elem c:A)
  If (((a<=b) && (b<=c)
    && !(a<=c))) ok = false;

If (ok)
  cout << "Transitive" <<
       endl;
```

By default, $\mathbb{A}$ is equipped with a **linear order** $\leq$, isomorphic to the rational numbers. We test that the order is transitive, by setting a **boolean** flag of type `lbool` whenever a counterexample is found. Note that we use a special **conditional** `If` with LOIS, actually a macro. This outputs *Transitive*.

Now we demonstrate an algorithm manipulating on infinite sets, namely the reachability algorithm for infinite graphs. Reachability is very important for applications of LOIS [KTa].

```
lsetof<elpair> E;

for(elpair x: Pairs)
  for (elpair y: Pairs)
    If ((x.second == y.first)
      && ((y.second != x.
          first)))
      E += elpair(x,y);

cout << E << endl;
```

First, let's define some infinite directed graph. In this case, the vertices are pairs of elements of A, and the edgeset is E. This outputs $\{((p,q),(q,r))|r{\in}\mathbb{A},\ q{\in}\mathbb{A},\ p{\in}\mathbb{A},\ r{\neq}p\}$. Therefore, edges are of the form $(p,q) \to (q,r)$, where $p \neq r$.

```
lbool reached = false;

for (elem a: A)
  for (elem b:A) {
    elpair s = elpair(a,b);
    elpair t = elpair(b,a);
    If (memberof(t,reach(E,
        newSet(s))))
      reached = true;
}

If (reached) cout << "Reached
    " << endl;
```

For each pair (a, b) of distinct elements of A, we test reachability of (b, a) from (a, b) in our graph, using the function `reach` given below (and defined outside of the body of the main function). Note that we use **local variables** s and t. The output is *Reached*. An example (shortest) path is $(a, b) \to (b, c) \to (c, d) \to (d, b) \to (b, a)$, where c, d are distinct from a, b.

```
rset reach
  (rsetof<elpair> E,
    rset S) {
  lset R = S;
  lset P;
  While (P!=R) {
    P = R;
    for (elpair e: E)
      If (memberof(e.first, R
          ))
        R += e.second;
  }
  return R;
}
```

We use a **fixpoint algorithm** to compute the vertices reachable from S in a graph with edge-set E. Special **while loops** are used with LOIS conditionals (again, macros). Correctness of this algorithm is clear. One could also use a BFS traversal, however, the presented algorithm is slightly easier to analyse. Termination (in finite time) can be proved for *any* graph defined in LOIS from the domain $\mathbb{A}$, using equality only [KTa].

The intuitive semantics of the above examples is clear, as they follow closely the set-builder notation – we iterate through some set and collect the results in some resulting set (an exception the use of the boolean flag). In general, our novel *pseudo-parallel* semantics is meaningful also when other operations are performed inside the loops, e.g. removing elements from a set, or declaring local variables within the body of the loop. This requires extending the set-builder intuitions, and defining a proper semantics [KTb].

3 Overview

This section gives an overview on the main features of LOIS, and how they have been implemented, and how to use them. More details will be given in the further sections.

LOIS is based on the hybrid pseudoparallel semantics. The paper [KTb] explains this semantics in detail, and how an interpreter can be implemented in practice by using *contexts*, which form a stack which changes whenever a local variable is created (possibly by a `for` loop), or a conditional is used. C++ allows a programming technique known as RAII, that is, automatic initialization and finalization of variables when a local variable enters or exits the scope. This is exactly what we need – our implementation uses RAII to change the contents of the LOIS stack when **if**, **for**, and **while** constructs are used. Note that, since **if** and **while** change the current context, we were unable to use C++'s `if` and `while` statements directly – instead, `If` and `While` macros are used, and `Ife` for if-then-else.

The type `lset` represents a LOIS set together with its inner context, and the polymorphic type `elem` represents a v-element. V-elements can represent an integer (type `int`), a term over $\mathcal{A}$ (type `term`), a pair (type `elpair`), a tuple (type `eltuple`), or a set (type `lset`). Integers, pairs, and tuples are implemented with the corresponding standard C++ types (`int`, `std::pair` and `std::vector`, respectively); and more types can be added by the programmer. So the programmer can for instance extend `elem` to allow a type representing lists or trees of elements, thus allowing sets of type `lset` to store infinite sets of lists or trees. It is well known that integers, pairs and tuples can be encoded in the set theory (using Kuratowski's definition of pair, for example); however, allowing to use them directly in our programs greatly improves both readability and efficiency.

Hybrid pseudoparallel looping over a set X of type `lset` is done with `for(elem x:X)`. This is implemented using the C++11 range-based loop. We can check the specific type of `x`, as well as inspect its components, with functions such as `is<T>`, `as<T>` (where `T` is one of the types listed above) and `isSet`, `asSet`. The syntactic sugar `lsetof<T>` is provided for defining sets which can only include elements of one specific type `T` – this allows static type checking, and eliminates the necessity of using `is` and `as` functions.

In some cases, such low-level representation of elements is not enough: for example, consider the function `extract(X)` which returns the only element of a set X of cardinality 1. If $X = \{a|a = b\} \cup \{0|a \neq b\}$, then it is not possible to represent `extract(X)` as `elem` in the context $\{a \in \mathbb{A}, b \in \mathbb{A}\}$, since each `elem` has to be of specific type, and in our case, `extract(X)` can be either a term or an integer. In this case, we can use the type `lelem`, which represents *piecewise v-elements* (see Section 6) – that is, ones which may have different representations depending on the constraints satisfied by variables in the context. Internally, this type is represented with a set – thus, `extract(X)` simply wraps the set X into a piecewise element. Type `lbool` represent a piecewise boolean variable, which boils down to a formula with free variables from its inner context.

All the conditions appearing in `If` and `While` statements are evaluated into first-order formulas over the underlying structure $\mathcal{A}$. A solver is used to check whether the set of all constraints on the stack is satisfiable (and thus whether to execute a statement or not). Also, a method of simplifying formulas is necessary, to obtain legible presentations of results, and to make the execution of the sequel of the program more effective.

The membership function `memberof(X,Y)`, as well as set equality `X==Y` and inclusion `subseteq(X,Y)`, have been implemented straightforwardly using `lbool` and hybrid pseudoparallel iteration over the sets involved. They are defined with a mutual recursion – set equality is a conjunction of two set inclusions, set inclusion $X \subseteq Y$ is evaluated by looping over all elements of X and checking whether they are members of Y, and membership $x \in Y$ is evaluated by looping over all elements of Y and checking whether they are equal to x. Equality and relation symbols applied to terms result in first order formulas.

Furthermore, for technical reasons, types `rbool`, `rset` and `relem` are used – these types are used for rvalues, while `lbool`, `lset` and `lelem` are used for lvalues.

This is because lvalues have an inner context, while rvalues do not (their inner context always equals the current context).

To enforce fully pseudoparallel computation, thus simulating LOIS^0 from [KTb], write

```
for(relem e: fullypseudoparallel(X)).
```

The underlying structure $\mathcal{A}$ is not given at the start of the program; instead, it is possible to define new sorts and new relations during the execution. Our prototype includes several relations with decidable theories (order, random partition, random graph, homogeneous tree), as well as solvers for these theories. Also, it allows consulting external SMT solvers.

4 Syntax and features

In this appendix, we list the key constructions available in LOIS. Also, we give an informal description of some constructs, which is sufficient for understanding basic programs in LOIS on an intuitive level. This should not be confused with the formal semantics, which is given in [KTb].

4.1 Types

LOIS allows the programmer to manipulate sets, which are represented by the types `lset` and `rset`. The type `rset` should be used for rvalues, i.e., for temporary values (such as values returned by functions or for function arguments) and `lset` should be used for lvalues, e.g. for local variables which are not function arguments. The reason why two different types are used for lvalues and rvalues is the pseudoparallel semantics [KTb]; lvalues have an inner context, while rvalues have not. Intuitively, a local value has a copy for each pseudoparallel thread that existed at the moment of its creation, while a temporary value has a copy for each pseudoparallel thread that exists at the moment while this value exists. The same convention (having a pair of types for lvalues and rvalues) is also used for other types below.

There is a polymorphic type `elem` representing elements of sets, which should be used as the type of a control variable in a `for` loop. The programmer can define set elements basing on any C++ type which has the following basic operations defined: variable substitution (create a copy with the variable changed), output (to a C++ stream), equality test, and checking whether it depends on a given variable (required for optimization). Integers, pairs (type `elpair`), and tuples (type `eltuple`) are based on the standard C++ types, and are defined as `int`, `pair<elem, elem>` and `vector<elem>`, respectively. Terms are represented by the type `term`.

The variants `lelem` and `relem` represent elements of a set. While `elem` contain elements of a fixed type (e.g., a pair), `lelem` and `relem` represent *piecewise elements*, whose type may depend on the variables in the context (current for `relem`, inner for `lelem`) – intuitively, their type may different in different pseudoparallel threads. They are described in detail in Section 6 below. Also, `lnum<T>` and `rnum<T>` are wrappers for piecewise numbers – they are described in Subsection 6.1 below.

Finally, there are two types for representing booleans whose values may depend on variables in their current (inner) context, `lbool` and `rbool`. They are internally represented as first-order formulas depending on variables in their context (inner for lbool, current for rbool).

Conversions. There is an assignment operator for assigning a variable of type `relem` to a variable of type `lelem`. Thanks to this and the C++ conversion mechanism, whenever a variable of type `lelem` is used in place where a variable of type `relem` is expected, an automatic conversion is performed by the compiler (and similarly for the other pairs of lvalue and rvalue types). To convert a `relem e` to a `elem`, `for(elem x: newSet(e))` can be used. Also, pairs, tuples, terms, integers, and sets are automatically converted to `elem`s. A variable `x` of type `elem` can be cast to a set, a pair, a tuple, a term, or an integer, using the operations `asSet(x)`, `as<elpair>(x)`, `as<eltuple>(x)`, `as<term>(x)`, and `as<int>(x)`, respectively. It can be also tested whether `x` is a pair, by the operation `is<elpair>(x)` (and similarly for terms, tuples and integers) which returns a value of type `bool`.

Sets with type checking. The types `lsetof<T>` and `rsetof<T>` are used instead of `lset` and `rset` to create sets whose all elements are of type `T` (typically, `term`, `elpair`, `eltuple`, `rset`, `int`, or `rsetof<U>`). This is a wrapper around `lset` and `rset` which allows only adding elements of type `T`; moreover, the `for` loop iterates over type `T` instead of the polymorphic `elem`. This improves readability and provides type checking.

4.2 Flow control

LOIS has the following constructs constructs for flow control:

- The conditional `If (cond) I`, where `cond` is a condition of type `rbool` and `I` is an instruction (the body) which is to be executed if the condition is satisfied. There is also a variant `Ife (cond) I else J`, where the instruction `J` is to be executed if the condition fails. `If` and `While` are macros; they should be used with LOIS conditions, rather than the normal **if** and **while**.
- The looping construct `While (cond) I`, where `cond` and `I` are as above.
- The *hybrid pseudoparallel* looping construct `for (elem x:X) I`, where `x` is the name of the introduced control variable, `X` is the set (of type `rset`) over which it ranges, and `I` is an instruction (the body of the loop).
- The fully pseudoparallel looping can be achieved in LOIS using the construct `for(relem e: fullypseudoparallel(X))`.

Furthermore, functions and recursion from C++ can be used. Using the rvalue types (such as `rset` or `relem`) is recommended for arguments and return values, unless we are passing references to lvalues as arguments to the function, when the lvalue type should be used. Note that since the `for` loop in LOIS is defined in a hybrid way [KTb], using the `return`, `break`, or `continue` statement inside a loop will cause LOIS to stop processing the set, and thus unintuitive behavior. The recommended approach is to create a value (say, `lelem ret`) to represent the returned value, set its value in the loop, and then `return ret`) after the loop ends.

4.3 Operations

The basic operations on sets (`rset`) are defined as follows. In the list, we denote sets (`rset`) with `X` and `Y`, elements (`relem`) with `x`, and set lvalues (`lset`) as `Z`.

Note that the assignment and compound operators (`= += |= &= &=~`) are defined according to the pseudo-parallel semantics [KTb]. That means, for each valuation v of the inner context of Z, the operation is performed on Z_v in parallel for each valuation w of the current context which extends v. Also note that adding elements to a set is much more efficient than removing (which basically loops over the set and keeps only the elements which are not to be removed).

`X==Y`	set equality
`X!=Y`	set inequality
`X&Y`	set intersection
`X&~Y`	set difference
`X|Y`	set union
`X*Y`	Cartesian product (`elpair` used for the pairs)
`cartesian({X,Y,Z,...})`	Cartesian product (`eltuple` used for the tuples)
`subseteq(X,Y)`	`X` is a subset of `Y`
`memberof(X,x)`	`x` is an element of `X`
`newSet()`	create an empty set
`newSet(x)`	create the set {`x`}
`newSet(x,y)`	create the set {`x`, `y`}
`newSet({x,y,...})`	create a set with the given elements
`extract(X)`	extract the single element of a set
`Z=X`	set `Z` to `X`
`Z+=x`	add `x` to the set `Z`
`Z-=x`	remove `x` from the set `Z`
`Z|=X`	add all elements of `X` to the set `Z`
`Z&=X`	remove all elements of `Z` which are not elements of `X`
`Z&=~X`	remove all elements of `Z` which are elements of `X`

4.4 The underlying structure

To create an infinite set, construct an object of the class `Domain`; this object's method `getSet()` returns the underlying set, of type `rset`. It is possible to create multiple domains.

All domains are automatically equipped with equality (`==`, `!=`) and a dense total order, accessed with the usual operators `<`, `>`, `<=`, `>=`.

LOIS supports domains which have more structure, for example, with two independent dense total orders. This is explained in detail later (Section 5).

4.5 Quantifier macros

LOIS defines macros `FORALL`, `EXISTS`, `FILTER`, `MAP`, and `FILTERMAP`, allowing the programmer to construct formulas with quantifiers, and sets, intuitively. `FILTER`$(x, A, \phi(x))$ corresponds to the mathematical set $\{x|x \in A, \phi(x)\}$, `MAP`$(x, A, v(x))$ corresponds to $\{v(x)|x \in A\}$, and `FILTERMAP`$(x, A, \phi(x), v(x))$ corresponds to $\{v(x)|x \in A, \phi(x)\}$. All these macros are defined using the for loop, and are therefore redundant in terms of expressive power.

4.6 Displaying the current context, and naming variables

The variable `currentcontext` of type `contextptr` contains a pointer to the current context on the stack, while `emptycontext` is the pointer to an empty stack. (See [KTb] for the discussion of contexts and stacks.) Use `cout << c` to display the difference between the `currentcontext` and `c`. Thus, the following will display $|b > a|b{\in}\mathbb{A}|a{\in}\mathbb{A}|$:

```
contextptr c = currentcontext;

for(auto a: A) for(auto b: A) {

  as<term>(a).asVar()->name = "a";
  as<term>(b).asVar()->name = "b";

  If (a<b)
    cout << c << endl;
  }
```

Note how we have named the variables occuring in terms `a` and `b` to a and b – otherwise LOIS would not know how the programmer named them, so it would generate its own names, which would be probably some random letters instead of a and b.

Other functions working with contexts include `branchset(contextptr anccontext)` (which returns the set of all pseudoparallel threads since `anccontext`), and `getorbit` (see `loisextra.h`), which returns the orbit of a given element (see [KTa]), treating the variables in the given ancestor context as fixed.

4.7 Declaring atoms and axioms

One can declare atoms and axioms, like in the following:

```
declareatom u(&dA, "u"),
  v(&dA, "v");
axiom ax(u != v);
```

Create an object of type `declareatom` to name one element of the domain. Create an object of type `axiom` to add an axiom, for example, that the declared atoms are not equal (this is not given). This is implemented by pushing the respective variables and constraints on the stack. A careful reader will notice that the real effect is exactly the same as would be obtained by using `for(a1,A) for(a2,A) If(a1!=a2)`. However, when the programmer wants simply to select some elements of $\mathbb{A}$, these constructs are more intuitive than loops and conditionals.

4.8 Choosing the solver

The global variable `solver` of type `solveptr` describes the solver currently used by LOIS. The following solvers, or solver combinations, are available:

- `solverCrash()` which cannot solve anything (it just crashes).
- `solverBasic()` which can solve only the trivial cases.
- `solverExhaustive(int t, bool v)` calls the internal solver. The number `t` corresponds to the number of tries (possible valuations) after which the internal solver gives up, and `v` is the verbosity.
- `solverSMT()`, `solverCVC()` and `solverSPASS()` call the given external solver. They accept an optional `std::string` argument, which is the path to the solver executable.
- `solverIncremental(std::string)` uses Z3's incremental solving feature.
- `solverCompare(std::initializer_list<solveptr> p)` compares the results of two or more solvers, and checks for inconsistencies.
- `solverVerbose` and `solverNamed(std::string n, solveptr s)` are wrappers around solvers which provide extra diagnostic information.
- `solverStack(solveptr s, solveptr fallback)` calls the solver `s`, then calls `fallback` if it failed. It can be used, for example, to solve simple cases with the basic or exhaustive solver, then proceed to verbose exhaustive or external solver for the harder cases.

Call `useDefaultSolver(int i, int j)` to use the default solver (queries with complexity below i are solved in a non-verbose way, then they are solved in a verbose way, and the internal solver gives up at j tries; also j is used as the limit for the number of tries for the simplification algorithm).

4.9 Other functions

Some other functions include:

- `rset optimize(rset x)`, which optimizes the set by removing repetitions (each element in the returned set will be in exactly one set-builder expression).
- `rset optimizeType(rset x, rset type)`, which is a more efficient version of `optimize` for the case when we know that `x` is a subset of a (simple) set `type`.
- `getsingletonset(rset X)`, which returns the set of singletons of elements of X, represented as a single set-builder expression.

5 Domains, symbols, and relations

It is possible to equip domains with extra structure, such as an order. LOIS includes an internal solver for homogeneous with extension bounds [KTa]; also, several external solvers can be used for the structure $(\mathbb{N}, +)$ or $(\mathbb{R}, +, *)$. Such an extra structure does not have to be immediately declared when the domain is created; instead, at any time the programmer can create an object from one of the subclasses of the class `Relation`, for example with the following declaration:

```
RelOrder newOrder(" GT ", " LEQ ", " MAX ", " MIN ");
```

Terms representing elements of any domain can now be compared with respect to `newOrder` (a dense total order without endpoints) using the methods `rbool less(const term& a, const term& b)` or `rbool leq(const term& a, const term& b)`. We can also find the maximum and minimum of two elements, using the methods `term max (const term& a, const term& b)` and `term min (const term& a, const term& b)`.

When the formulas are displayed on the screen, one of the symbols given in the declaration of `newOrder` is used. Symbols are given as objects of class `symbol`; a conversion from `const char*` and `std::string` to `symbol` is provided, but the class `symbol` also allows to use symbols which are displayed differently depending on the context. For example, the object `sym` contains many symbols which are either used by LOIS itself or considered useful in applications, and methods `useUnicode()`, `useLaTeX()`, and `useASCII()`, which sets all the symbols to use the given format. The following symbols are defined in the object `sym`:

– Logical symbols: `exists`, `forall`, `_and`, `_or`, `eq`, `neq`, `in`
– Basic set symbols: `emptyset`, `ssunion` (an union of set-builder expessions), `leftbrace`, `rightbrace`, `sfepipe` (a separator between the value and the context in the set-builder expression), `sfecomma` (a separator between in the context of a set-builder expression), `pseudo` (operator which extracts a single element from a set, for the purpose of displaying `lelems`)
– Relational and functional symbols: `leq`, `geq`, `greater`, `less`, `max`, `min`, `plus`, `times`, `minus`, `divide`, `edge`, `noedge`, `arrow`, `noarrow`

LOIS declares the *main order*, which is defined with the following line:

```
mainOrder = new RelOrder(sym.greater, sym.leq, sym.max, sym.min);
```

The C++ opperators `<`, `>`, `<=` and `>=`, when used on terms, are defined as referring to `mainOrder`. When two orders (say, `mainOrder` and `newOrder`) are defined and used in the same formula and over the same domain, they are considered to be unrelated – intuitively, two random orders have been independently chosen over our domain. In general, the same rule of independence is used when defining multiple relations; in many cases, this yields oligomorphic homogeneous structures, which still have a decidable first order theory [KTa]. The algorithm used by LOIS for deciding the satisfiability only takes into account relations which are actually used in the given formula; so, the existence of `mainOrder`, even if it is not used, won't make the computations slower.

The following subclasses of `Relation` are available:

– `RelOrder`, explained above. The constructor of `RelOrder` has four symbol arguments: `greater`, `leq`, `max`, and `min`. `RelOrder` has four methods: `less`, `leq`, `min`, and `max`.
– `RelBinary`, which creates a random binary relation over the domain. The constructor of `RelOrder` has two symbol arguments `inrel` (in the relation) and `notinrel` (not in the relation), as well as two parameter `l` and

s, which define the properties of relation to be created. The parameter l can take the value of lmNoLoops (irreflexive), lmAllLoops (reflexive), or lmPossibleLoops (a can be in the relation with itself or not). The parameter s can take the value of smSymmetric, smAsymmetric, or smAntisymmetric. In all cases, with probability 1 we get the same graph (up to isomorphism), which has a decidable first order theory due to being homogeneous and oligomorphic (see [KTa] for details); when choices which leave more possibilities (lmPossibleLoops, smAsymmetric) are chosen, the internal solver has to consider all of them to verify the satisfiability of formulas. The relation is accessed with `rbool operator () (const term& a, const term& b)`.

- RelUnary, which creates a random partition of the domain. The constructor is `RelUnary(symbol rel, int n)`, where every element of the domain is randomly assigned to one of the n parts. Operator `rbool operator () (const term& a, int v)` is used to check whether a is in the part number v (0-based), and for convenience, the method `rbool together(const term& a, const term& b)` checks whether a and b are in the same parts.
- RelTree, which defines a homogeneous tree structure. This is an infinite tree, where every two elements have the least common ancestor, and it is dense without endpoints, that is, if u is an ancestor of v, then there is a w such that w is ancestor of v and u is ancestor of w, and there is an ancestor of u and a descendant of w; furthermore, there is infinite branching, that is, if $v_1, \ldots, v_k$ are descendants of u, then there is v such that the least common ancestor of v and v_i for $i = 1, \ldots, k$ is u. This structure showcases the fact that, in a homogeneous structure with extension bound, the isomorphism type of $\{a_1, \ldots, a_n\}$ might be defined not only by relations on these elements – we also need to check the relations of terms (in this case, using lca). Methods anceq ("ancestor or equal") and lca are used to access the relations and functional symbols. The constructor has three symbol arguments opanceq, opnotanceq, and oplca, which correspond to the methods.
- RelInt and RelReal, which define the set of integers and reals, respectively. The constructor defines the symbols related to the order, and also opplus, optimes, opminus, and opdivide. The term `constant(Domain *d, int i)` is used for integer constants, and there are also methods plus, times, minus, and divide for the basic operations. Note that these are not ω-categorical, and thus they are not compatible with other relations, and the internal solver does not work with them – an external solver is required [KTa]. Use the namespace orderedfield_ops from loisextra.h to conveniently bind the C++ operators (`+ - * /`) to methods of the given RelInt or RelReal.

Note that the domain for a given term can be obtained with the method `getDom()`.

6 Piecewise v-elements

The types lelem and relem are useful for representing variables which behave in different ways under different valuations, e.g., $1|x = y;\ x|x \neq y$.

For a formal syntax of those types, we need the following notions. A *piecewise* v-element is an expression e of the form $e_1|C_1; e_2|C_2; \ldots; e_k|C_k$, where $e_1, \ldots, e_k$ are v-elements, and $C_1, \ldots, C_k$ are constraints (i.e., first order formulas). If v is a valuation of the free variables of e, then $e[v]$ is defined as $e_i[v]$, where $1 \leq i \leq k$ is such that v satisfies C_i. If no such i exists, or it is not unique, then $e[v]$ is undefined.

The types `lelem` and `relem` represent piecewise v-elements. Similarly to `lset`, `lelem` is associated with an inner context.

Assignments to `lelem`. In [KTb] we only described how the assignments to variables of type `lset` are carried out. A variable `x` of type `lelem` designates a piecewise v-element. To see why using piecewise v-elements is necessary, consider the following example.

```
lset Z;
for (elem x:A) for (elem y:A) {
  lelem u = x;
  If (x != y)
    u = newSet(elpair(x,y));
  Z += u;
}
```

In order to guarantee the appropriate value of `Z` after executing this code, at the moment of the instruction `Z+=u`, the variable `u` needs to designate the piecewise v-element $x|x = y; (x, y)|x \neq y$.

Internally, LOIS represents a variable `x` of type `lelem` designating the piecewise v-element $e_1|C_1; e_2|C_2; \ldots; e_k|C_k$ by a variable ⟨`x`⟩ of type `lset`, which designates the v-set $\{e_1|C_1\} \cup \{e_2|C_2\} \ldots \cup \{e_k|C_k\}$. The v-set ⟨`x`⟩ can be obtained in LOIS by the instruction `newSet(x)`. Operations on the type `lelem` are carried out by lifting them to operations on the type `lset`. For example, the assignment `x = y` to the variable `x` of type `lelem` is simulated by executing the assignment ⟨`x`⟩ `= newSet(y)`.

New set. If `x` is of type `elem`, then the instruction `newSet(x)` is executed as expected– it constructs a v-set of the form $\{x\}$. If `x` is of type `lelem`, this operation becomes slightly more involved, as described in the paragraph "assignment" above.

Extract. If `X` is of type `rset`, then the result of `extract(X)` is of type `relem` and designates the piecewise v-element corresponding to the v-set X. On the level of implementation, the return value `u` of this instruction is such that ⟨`u`⟩=`X`.

Example 1. To see the usefulness of this operation, consider the following function which calculates the maximum of a set of terms, such as $\{a_1, a_2\}$ or $\{x \mid a_1 \leq x \leq a_2\}$. We assume here that domain $\mathbb{A}$ is equipped with a linear order $\geq$.

```
lelem max(rset X) {
  lset answer;
  for(elem x: X) If (FORALL(y, X, x >= y)) answer += x;
  return extract(answer);
}
```

First, we loop over all elements $x \in X$, and if $\forall y \in X. x \geq y$, add x to the set *answer*. If the set X indeed has a maximum, then *answer* will be a singleton containing this maximum; otherwise; *answer* will be empty. If X is equal to v-set $\{a_1, a_2\}$, then *answer* will be calculated as $\{a_1|a_1 \geq a_2; a_2|a_2 \geq a_1\}$ and the instruction `extract(X)` will return the piecewise v-element $a_1|a_1 \geq a_2; a_2|a_2 \geq a_1$.

6.1 Piecewise numbers

The types `lnum<T>` and `rnum<T>` (where `T` is usually `int`, but could be extended to other types) represent piecewise numbers. Operators are defined for `lnum<T>` and `rnum<T>` according to the pseudo-parallel semantics; in particular, `x++` for each valuation v of the internal context increments $x[v]$ by the cardinality of the set of possible valuations of the current context which extend v. For example, consider the following program:

```
for(auto a: A) for(auto b: A) {

  as<term>(a).asVar()->name = "a";
  as<term>(b).asVar()->name = "b";

  lnum<int> i = 0;

  for(auto x: A)
    If (x == a || x == b)
      i++;

  cout << i << endl;
  }
```

If we know that $a \neq b$, this will output 2. If we don't know this, a representation of the piecewise number will be generated, which currently is

$$\sqsupseteq \{2|b{\neq}a;\ 1|\exists w{\in}\mathbb{A}\forall x{\in}\mathbb{A} x = w{\vee}(x{\neq}a{\wedge}x{\neq}b)\}\,.$$

Note the simplification algorithm failed to simplify the formula for the case when $a = b$.

7 Safety of programming

It is possible to create a LOIS program which compiles and terminates without a run-time exception, but nonetheless works incorrectly; for example, the following function will count the cardinality of the set incorrectly, due to using the type `int` instead of the piecewise integer `lnum<int>`.

```
int cardinalityBad(elem X) {
  int result = 0;
  for(elem x: X) result++;
  return result;
}
```

Of course, this problem cannot be completely solved—it is possible to create an incorrect C++ program even without LOIS. However, the programmer should be mostly safe, if they observe the following rules.

– The lvalue types (`lbool`, `lset`, `lnum`, etc.) are used for all the local (and global) variables, except the iterators for looping over sets and `setof<T>`'s, for which `elem` and `T` (or `auto`) are used respectively, and temporary values (function parameters, function return values), for which rvalue types may (and should) be used. The difference between lvalue and rvalue types is technical—to handle assignments and other changes correctly, lvalue types require extra information (the inner context), while rvalue types do not. We have decided to use two distinct types—the rationale is that a lvalue variable should correspond to a specific stack (context), and the stack changes during a function call or return; also avoiding this extra information could improve the efficiency. Assigning a rvalue to a lvalue (or, in general, modifying a lvalue in any way which takes a rvalue as a parameter) is legal only if the rvalue does not depend on the sort variables which have been introduced since the lvalue was created. Thus, the following implementation of `max` would throw an exception for some sets X:

```
relem maxBad(rset X) {
  lelem ans;
  for(elem x: X)
    If (ans.isUndefined() || x > ans)
      ans = x;
  return ans;
}
```

– Non-structural programming constructs, such as `continue`, `break`, `return`, and `goto` are considered harmful and should be avoided. Suppose a function is called with the current context C, then new constaints are pushed on the stack, changing the current context to C'. Now, the `return` statement is used. This will break the execution flow not only for all the C'-valuations, but actually for all the C-valuations, irregardless of whether they are C'-valuations or not (the necessary condition for executing the return statement is that Val(C') is non-empty). Moreover, `If` statements could execute both branches, and they are internally implemented with a `for` loop, which might make the behavior of `continue` and `break` different than expected. Thus, the only way of using the `return` statement which is not considered harmful is to declare a local lvalue as the first statement (before any loops), modify it in the body of the function, and then return its value as the last statement of the function. Note that this approach is similar to the one used in original Pascal.

8 Contents of the package

The LOIS package included contains the following:

– source code (LOIS library itself, and the sample programs described above).
– a Makefile which includes the `all` target which compiles the LOIS library and the two programs, runs the two programs, and saves the output.
– The result of `make all` (both binaries and their output), obtained on the first system.

8.1 Sample programs

The subdirectory `tests` includes some sample programs.

Tutorial The program `tests/tutorial.cpp` includes the tutorial given in Section 2, as well as some other code snippets quoted in this paper. These snippets, as well as their outputs, are inserted directly from `tests/tutorial.cpp` or its output.

Automatic tests The program `tests/autotest.cpp` performs some automatic testing of LOIS. This includes some interesting applications of LOIS, allowing one to see that LOIS runs correctly, and how fast does it run. The following tests are conducted:

– `testRandomBipartite`
  Let R be a random symmetric and anti-reflexive relation, and let A_1 and A_2 be a random 2-partition. Let $S(x, y)$ iff x and y are in different parts of the partition. We construct a new relation $E = R \cap S$. In graph theoretic terms, R is Rado's random graph, S is a complete bipartite graph, and E is a random bipartite graph. We take one vertex $x \in \mathbb{A}$ and run BFS on the graph, and ask about the number of iterations after which we have reached every vertex. The program correctly answers that every vertex is reached after 3 iterations.
  This test evaluates in roughly 3 milliseconds on the machine used for tests.
– `testTree`
  A function is given elements $x_1, \ldots, x_k$ of the homogeneous tree, and asks questions about relationships between them. Once the answers uniquely determine the substructure generated by $x_1, \ldots, x_k$, the substructure is presented in a readable form. For four elements without any relations, 416 possible structures are generated (262 if we know that all the four elements are not equal — see sequences A005264 and A005172 in [OEI]).
  This test evaluates in roughly four seconds on our machine (for four elements). This time is relatively long because of two reasons:
  - The extension bound of the homogeneous tree is relatively large ($e(n) = 8n - 4$, which gives the evaluation time of roughly $8^k k!$ according to Proposition 1 in [KTa]). In fact Proposition 1 is not optimal, all trees are generated in time roughly linear in the number of all trees, which is 416 for four elements. Still, it grows quite fast.
  - The program is very ineffective: currently, each question tries to generate all the possible structures from the beginning, even if we know that some possible structures have been already ruled out.

  Therefore, the running time is actually at least quadratic in the number of possible trees. This should be optimized in the future versions of LOIS.
– `testOrder`
  This test the basic properties of the order relation, and evaluates very quickly.

- `testAssigment`
  This checks whether an assignment exception is correctly thrown when we try to assign a value (`rbool`, in this case) which uses variables which are not in the internal context of the variable we are assigning to.
- `testQueue`
  This checks whether the `setof`'s and the queue semantics of the `for` loop works correctly. Numbers from 0 to 10 are inserted to `lsetof<int>`.
- `testRemoval`
  This checks whether the `-=` operator works in the natural, pseudo-parallel way, as advertised in paper.

Minimisation of an automaton The program `tests/mintest.cpp` tries to perform the minimisation algorithm on an orbit finite automaton. This automaton over the alphabet $\mathbb{A}$ (our infinite domain) reads three symbols, and accepts iff either two of them are equal (if there are less than three or more than three symbols, the word is rejected). The minimisation algorithm works in a way similar to the usual one for finite automata. The equivalence relation $\eta \subseteq Q \times Q$ is computed – two states will be in η if they can be merged into a single state. Initially, η is set to $F \times F \cup (Q - F) \times (Q - F)$, and then, in each iteration states each $x, y \in Q$ are separated iff $\neg\eta(\delta(x, a), \delta(y, a))$ for some symbol a in the alphabet. For this particular automaton minimisation takes four iterations. The algorithm is implemented using two representations (η is represented either as a relation or as the set of equivalence classes), and currently takes 0.15 s in the relation representation, and 32 s in the equivalence class representation.

Solver tests The program `tests/soltest.cpp` tests various solvers on several LOIS functions. The table in [KTa] is based on its results. The following tests are included:

- `testOrder` This test the basic properties of the order relation, and evaluates very quickly with the internal solver, although external solvers have problems with it.
- `testReachable` Reachability from the article (Section 2).
- `testReal` This test the basic properties of the `Real` sort (LRA logic).
- `testMinimize??` These tests minimize automata. There are two automata: A (the same as in `tests/mintest.cpp`) and B (the automaton using the integers from the introduction), and three different implementations of the minimisation algorithm (two from `tests/mintest.cpp`, and the one shown in the Introduction is implementation number 3). Internal solver and SPASS work on the automaton A, but none of the solvers work on B.
- `testPacking` What are the maximal sets of subsets of (0,5) such that no two points are in distance less than 1? Z3 correctly calculates using the LRA logic that such maximal sets can have from 3 to 5 elements.
- `testCirclePacking` This tests the NRA logic by asking about packings of disks in a larger disk. None of the tested solvers can answer even the simplest questions here.

The external solvers where called using the following commands:

Z3: `z3-*/bin/z3 -smt2 -in -t:500`
CVC4: `cvc4 --lang smt --incremental --tlimit-per=500`
CVC4*: `cvc4 --lang smt --incremental --finite-model-find --tlimit-per=500`
SPASS: `SPASS -TimeLimit=1`

References

[KTa] Eryk Kopczyński and Szymon Toruńczyk. *LOIS: an application of SMT solvers.* Submitted for a conference. See `http://www.mimuw.edu.pl/~erykk/lois/`.

[KTb] Eryk Kopczyński and Szymon Toruńczyk. *LOIS: syntax and semantics.* Submitted for a conference. See `http://www.mimuw.edu.pl/~erykk/lois/`.

[OEI] The on-line encyclopedia of integer sequences. http://oeis.org.